THE UNIVERSITY OF
MELBOURNE

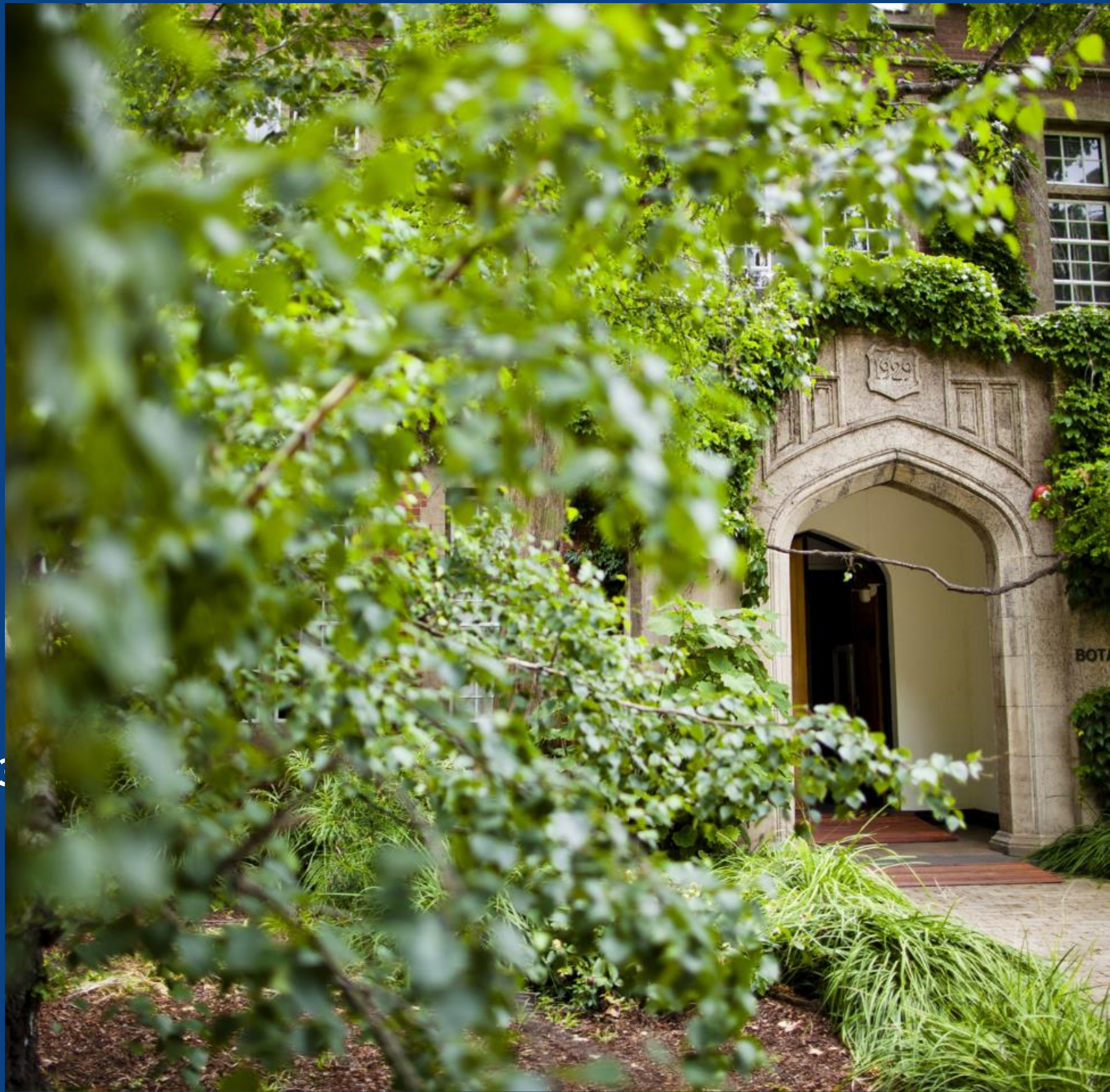
COMP 90048
Declarative Programming
Workshop 4 (week5)

2019 semester 1

by Wendy Zeng

Tutorial : Tue 18:15 - 19:15 221 Bouverie St, room B11

Wed 17:15 - 18:15 201 Bouverie St, room B132



Outline

1. Polymorphism and BST
2. Higher Order functions
   a. map and filter
   b. foldr and foldl
   c. zipWith

1. Polymorphism and BST

data Tree = Leaf | Node String Int Tree Tree

data Tree **k v** = Leaf | Node k v (Tree k v) (Tree k v)

data Tree **a** = Leaf | Node (Tree a) a (Tree a)

- k, v, a are **type variables that can represent any types**
- Is a type of **data abstraction**

1. Polymorphism and BST

- Tree sort:

List --------> BST --------> List

list_to_tree tree_to_list

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = quicksort smaller ++ [x] ++ quicksort larger
    where  smaller = filter (<pivot)   xs
           larger    = filter (>=pivot) xs
```

```
tree_to_list :: Tree a -> [a]
tree_to_list Leaf = []
tree_to_list (Node a l r) = smaller ++ [a] ++
larger
    where smaller = tree_to_list l
          larger   = tree_to_list r
```

2. Higher Order functions

- Take functions as argument or return a function as output

- map :: (a -> b) -> [a] -> [b]
- filter :: (a -> Bool) -> [a] -> [a]
- foldl :: (b -> a -> b) -> b -> [a] -> b
- foldr :: (a -> b -> b) -> b -> [a] -> b
- zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
- concatMap :: (a -> [b]) -> [a] -> [b]

2a. Map and Filter

- **map** :: (a -> b) -> [a] -> [b]
  - map _ [] = []
  - map f (x:xs) = f x : map f xs
  - Using list comprehension: [f x | x <- list]

- **filter** :: (a -> Bool) -> [a] -> [a]
  - filter _ [] = []
  - filter f (x:xs) = if f x then x : filter f xs else filter f xs
  - Using list comprehension: [x | x <- list, f x]

2a. Map and Filter

- **map** :: (a -> b) -> [a] -> [b]
  - map _ [] = []
  - map f (x:xs) = f x : map f xs
  - Using list comprehension: [f x | x <- list]
  - Using foldr: foldr ((:).f) []

- **filter** :: (a -> Bool) -> [a] -> [a]
  - filter _ [] = []
  - filter f (x:xs) = if f x then x : filter f xs else filter f xs
  - Using list comprehension: [x | x <- list, f x]
  - Using foldr: foldr (\x->if f x then (x:) else id) []

2a. Map and Filter

The term

```
map (:[]) "abc"
```

evaluates to which one of the following?

- ['a','b','c']
- ["abc"]
- "abc"
- $\boxed{\text{["a","b","c"]}}$
- *The term is in error because map applies only to lists, not to strings.*

2a. Map and Filter

Consider the function

```
inRange :: Ord a => (a,a) -> a -> Bool
inRange (lo,hi) x = x >= lo && x < hi
```

(Notice that the range is inclusive at the low end and exclusive at the high end. This is often a very convenient convention to observe when dealing with such intervals.)

Which one of the following terms correctly evaluates to

[1,2,3]

- filter ((1,7)`inRange`) [0,1,5,2,3,7] — **Correction: ((1,5) `inRange`)**
- filter (inRange (1,5) x) [0,1,5,2,3,7] — **Correction: (inRange (1,5))**
- filter (inRange (1,5)) [0,1,5,2,3,7]
- filter (\x -> inRange (0,4) x) [0,1,5,2,3,7] — **Correction: (\x -> inRange (1,5))**
- filter inRange (1,5) [0,1,5,2,3,7] — **Correction: (inRange (1,5))**

2b. Foldl and Foldr

- **foldl** :: (b -> a -> b) -> b -> [a] -> b
  - foldl _ b []   = b
  - foldl f b (x:xs) =

```
let newbase = f b x
in foldl f newbase xs
```

- **foldr** :: (a -> b -> b) -> b -> [a] -> b
  - foldr _ b []   = b
  - foldr f b (x:xs) = f x (foldr f b xs)

```
list_to_tree :: (Ord a) => [a] -> Tree a
list_to_tree [] = Leaf
list_to_tree (x:xs) = insert_to_bst x (list_to_tree xs)
```

↓

```
list_to_tree :: (Ord a) => [a] -> Tree a
list_to_tree = foldr insert_to_bst Leaf
```

2b. Foldl and Foldr

- A few more examples in foldr:

```
-- similar to what (++) does without traversing the
first list
effi_concat :: [a] -> [a] -> [a]
effi_concat left right = foldr (:) right left

-- Haskell implementation of reverse
reverse :: [a] -> [a]
reverse = foldl (flip (:)) []

-- replace each occurrence of element a with element b in a list
substitute :: Eq a => a -> a -> [a] -> [a]
substitute a b = foldr ((:).(\x -> if x==a then b else x)) []
```

Pointfree style

- write functions as a composition of other functions
- leave out the actual arguments applied on both side of the =

2c. zipWith

- zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
  - zipWith _ [] _ = []
  - zipWith _ _ [] = []
  - zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys

Using map:

```
transpose [] = []
transpose ([]:xss) = transpose xss
transpose ((x:xs):xss) = (x: head_lst xss) : transpose (xs : tail_lst
xss)
    where head_lst = map (\row -> head row)
          tail_lst  = map (\row -> tail row)
```

Using zipWith:

```
transpose [xs] = map (\x -> [x]) xs
transpose (xs:xss) = zipWith (:) xs (transpose xss)
```

THE UNIVERSITY OF MELBOURNE

Thank you

wendy.zeng@unimelb.edu.au

By Wendy Zeng